# Variable-Based Analysis for Traceability in QVT-R Model Transformations

Omar Martínez Grassi, Claudia Pons, Gabriel Baum

LIFIA, Facultad de Informática, UNLP, CAETI - Universidad Abierta Interamericana (UAI),
CIC (Comisión de Investigaciones Científicas de la Provincia de Bs.As.), La Plata, Bs. As, Argentina.

omartinez@rionegro.com.ar, cpons@lifia.info.unlp.edu.ar, gbaum@info.unlp.edu.ar

**Abstract** Model-driven software development promotes models as the primary artifacts in software development. During all development phases, different models of the system are created, refined, and linked, such that from the requirements to the implementation a whole network of models is built. OMG's "Model-driven Architecture" (MDA) is one important instance of this general paradigm, with a particular stress on automation of model creation and linkage. Models are complemented by model transformations written in the QVT language, describing how one model can be derived from another one. Traceability data in the MDA can be understood as the runtime footprint of model transformations. It has many applications: to perform change impact analysis, to keep consistency between models, to carry out requirement coverage analysis, etc. In this paper we present a proposal to enhance the retrieval of traceability information in MDA. The contributions of our work include the minimization of manual efforts to achieve traceability, as error-prone and time-consuming activity, and the attainment of independence regarding the QVT engine implementation.

## 1 Introduction

Model-Driven software development (MDSD) [7] promotes models as the primary artifacts in software development. During all development phases, different models of the system are created, refined, and linked, such that from the requirements to the implementation a whole network of models is built. OMG's "Model-driven Architecture" (MDA) [9] is one important instance of this general paradigm, with a particular stress on automation of model creation and linkage. Models are complemented by model transformations, describing how one model can be derived from another one. Thus, the linkage between models can be obtained automatically and repeatedly. A common basis for describing modeling and transformation languages is used: the "Meta-Object Facility" (MOF). For expressing model transformations, MDA provides the MOF-based "Query/View/Transformations" language (QVT) [10]; in this language, translations from models of one MOF-based modeling language into models of the same or another MOF-based modeling language can be specified.

The IEEE Standard Glossary of Software Engineering Terminology [12] defines traceability as the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another or the degree to which each element in a software development product establishes its reason for existing. This early definition is strongly influenced by the originators of traceability, i.e. requirements management community. However, it is possible to find a much broader one, more useful for the purposes of model-driven development. In [1], Aizenbud defines traceability as "any relationship that exists between artifacts involved in the software engineering life cycle". So traceability is achieved by defining and maintaining relationships between artifacts involved in the software-engineering life cycle during system development.

Traceability data in the model-driven development can be understood as the run-time footprint of model transformations [3]. Trace links can provide this information by associating source and target model elements with respect to the execution of a certain model transformation. Trace links have many applications [3]: performing impact analysis (i.e., analyzing how changing one model would affect other related models), determining the target of a transformation as in model synchronization, at model-based debugging (i.e., mapping the stepwise execution of an implementation back to its high-level model), and in debugging model transformations themselves. In the context of specific scenarios of MDSD, there are even more use cases for trace links [14]: supporting design decisions by consulting traceability links, providing validation by analyzing traceability links transitively through the different models to analyze if a requirement is fulfilled and if an artifact is up to date, understanding and managing MDSD artifacts by the many dependencies that exist between them, understanding and debugging transformations, and carrying out model synchronization. Furthermore, trace links can be useful to implement change impact analysis, by simulating changes to an artifact and exploring their effects on other artifacts. Finally, traceability information can be used both in coverage analysis, to determine whether all requirements were covered by test cases in the development life cycle, as in orphan analysis, to find orphaned model elements with respect to an specific trace link.

Model transformation approaches allow either implicit or explicit trace link generation [3]. In the former case, dedicated support for tracing is provided, while in the latter one, developers have to manually encode the creation of traceability links in the transformation rules, as a regular output model. Although QVT provides dedicated support for tracing, it has significant limitations [2]. Explicit trace link generation must rely on human effort, an error-prone and time-consuming activity. In this paper we propose a schema for traceability support in QVT model transformations, based on QVT code analysis, which allows the inference of traceability information between the source and target models from the model transformation specification, systematically, without requiring any additional code or intervention from the developer.

The remainder of this paper is structured as follows: in Section 2 we describe the background and motivation of our work. Then in Section 3 we present our proposal for obtaining traceability information automatically. In Section 4 we evaluate the proposal and make a comparison to related works. Finally, in Section 6, we present our conclusions and future works.

## 2 Background and motivation

Some years ago, the OMG adopted QVT (Query/View/Transformation) language as a standard for model transformation specification. QVT is a hybrid declarative/imperative language [10], which integrates the standard OCL 2.0 and extends its imperative version, defining three specific domain languages (DSL) called Relations, Core (both declarative) and Operational Mappings (imperative). The QVT standard describes a trace which is recorded all along the transformation execution, and can be serialized at the end of the transformation execution in order to be inspected later. However, its main purpose is to support the object resolution mechanism, indispensable to perform the transformation. The trace provided by QVT is thus mainly internally used by the engine and is not well adapted to the traceability purposes. In [2] several scenarios are described where the basic information required by most activities based on trace exploitation is not provided by the QVT trace. Moreover, the information contained in the trace is relative to some kind of elements (*Class* and not *Property*). Thus, trace links are only one-to-one links between elements of the root pattern of the rules (i.e., elements defined at rule's signature).

To overcome these limitations we propose a different novel approach: to get traceability information directly from the transformation program, i.e., the QVT code. While there are several approaches that allow getting traceability information implicitly, what is new in the mechanism we propose here is the possibility to obtain not only trace links between source and model elements, but the traceability relationships as well. The traceability relationships are those that are defined between types (e.g., metamodel elements); trace links are instances of these relationships (e.g., between model elements). While the trace links allow us to know the origin of target model elements in a transformation of a given instance of source and target models, the traceability relationships allow us to predict the trace links for every source and target models submitted to this transformation.

Essentially, our goal is to develop a solution to enhance the attainment of traceability information in QVT. The contribution of our work includes minimizing the manual efforts to achieve traceability, as error-prone and time-consuming activity, and obtaining independence regarding the QVT engine implementation.

## 3 Variable-based analysis

This work addresses the problem of obtaining traceability information automatically, i.e. without having to depend on someone to specify how the trace information will be obtained. Our work shows that inferring traceability relationships by analyzing the QVT model transformation source code is a feasible task. This analysis is based on the recognition of certain patterns in the use of variables within the specification of a model transformation written in QVT Relations language (QVT-R), which can be used to discover trace links between source and target model elements. We call this approach “variable-based analysis”.

In order to understand our proposal some QVT language background is necessary. In the first part of this section we will introduce the reader to some of the basic concepts of the QVT-R language.

## 3.1 The QVT Relations language (QVT-R)

In the QVT-R language, a transformation between candidate models is specified as a set of relations (or rules) that must hold for the transformation to be successful. A candidate model is any model that conforms to a model type. A transformation invoked for enforcement is executed in a particular direction by selecting one of the candidate models as the target. The target model may by empty, or may contain existing model elements to be related by the transformation. The execution of the transformation proceeds by first checking whether the relations holds, and for relations for which the check fails, attempting to make the relations hold by creating, deleting, or modifying only the target model, thus enforcing the relationship [10].

Every relation in a transformation declares constraints that must be satisfied by the elements of the models. A relation is defined by two or more domains and a pair of when and where predicates. A domain is a typed variable that can be matched in a model of a given model type. The when clause specifies the condition under which the relationship needs to hold, while the where clause specifies the condition that must be satisfied by all model elements participating in the relation. A relation in a QVT program can be a top-level relation or a non-top-level relation. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the where clause of another relation.

Domains can be marked as checkonly or enforce. When a transformation is executed in the direction of a checkonly domain, the existence of a valid match satisfying the relationship is simply checked. On the other hand, when a transformation executes in the direction of an enforced domain, if the check fails, the target model is modified so as to satisfy the relationship [10].

## 3.2 Variable-based analysis concepts

Variables in a QVT transformation play a major role in the matching mechanism at the time of performing the transformation from one model (source) to another model (target). Essentially, the variable-based analysis (VBA) proposed in this paper consists in the identification of each variable in every rule of a transformation, and the determination of the role it plays within the statement and the context where it is used.

Indeed, we have observed that certain patterns of variables usage within a QVT-R program allow us determining the relationships between the source and target model elements and therefore inferring the trace relationships between them. The VBA is based on the recognition of these patterns within a QVT-R program, and the subsequent building of the traceability information.

The basic steps that conforms the variable-based analysis are the following: 1) code inspection, 2) pattern matching process and 3) generation of trace links. The flow proposed by VBA is straightforward: the QVT-R code inspection is performed to find out one or more pattern-matching instances. If this happens, the traces are generated.

M1
+name : String
M2
+name : String

**(a)** MM1 Metamodel **(b)** MM2 Metamodel

**Fig. 1.** MM1 and MM2 metamodels

We present a simple scenario to illustrate how these "patterns" can help us to find the relationships between candidate models in a QVT-R transformation. Let MM1 and MM2 be two metamodels with a single class each one, named M1 and M2 respectively, which in turn contains a single name attribute (Figure 1), and consider a transformation from MM1 to MM2 that translates every instance of M1 into a new instance of M2 with the same name (Figure 2a).

```
transformation MM1_to_MM2(m1:MM1,m2:MM2) {
  top relation classM1_to_classM2 {
    cname : String;
    checkonly domain m1 a : MM1::M1  {
      name = cname
    };
    enforce domain m2 b : MM2::M2  {
      name = cname
    };
  }
}
```

**(a)** *MM1_to_MM2* trasformation

```
transformation T .. {
  top relation R {
    x : VarType;
    checkonly domain source {
      s = x
    };
    enforce domain target {
      t = x
    };
  }
}
```

**(b)** Pattern No.1

**Fig. 2.** QVT-R transformation code and Pattern No.1

The transformation contains a single mandatory (top-level) relation, *classM1_to_classM2*, which defines the following constraint: for every instance of M1, whatever its name, there must exists an M2 instance with the same name. As you can see, the *cname* variable acts as a pivot between both model elements "name" of M1 and M2, determining the constraint of the relationship. This, in turn, allows us to obtain traceability information: in the context of the *MM1_to_MM2* transformation, the name of every M2 instance is defined by the name of the M1 instance. We can be sure of that because this relationship is enforced: when the transformation runs in the direction of the model of the enforced domain, if check fails, the target is modified to satisfy the relationship.

To describe the conditions that allow us to get the traceability information from a relation we have chosen the word "pattern". In the context of our work, a trace pattern stated a consistent, characteristic usage of variables within a QVT program that leads to traceability data attainment. It is a textual description that includes the conditions which allow obtaining the traceability relationship between candidate model elements. We have identified six patterns that can be found in QVT-R programs. In the following section we will describe them in detail.

### 3.3 Trace patterns

The first pattern we observed is based in the usage of a variable as a pivot between two source and target model elements, as we described in the example above.

**Pattern No.1** The use of a variable in both checkonly and enforce domains.

When a top-level rule or a non-top-level rule invoked from a statement of a when, or where, clause of a top-level rule, assigns a value to a target model element *t* defined in the scope of an enforce domain, by using a variable previously used on a source model element *s* defined in a checkonly domain, then we say that the source model element *s* will map directly to the target model element *t*. This leads to the derivation of the trace relationship T: s→t. The operator → stands for "determine", meaning that the value of element *s* will determine the value of element *t*. In other words, the value of element *t* depends on the value of element *s* at the moment of perform the transformation.

Following our example of Figure 2a, we can see that variable *cname* is used in both checkonly and enforce domains. The relation *classM1_to_classM2* will be satisfied only when there is an M1 instance with the same name of an M2 instance, so we can get the trace M1::name→M2::name. This result says that for every M1 and M2 instances satisfying the relation *classM1_to_classM2*, the name of M2 will be exactly the same as the name of M1. The Figure 2b describes our first pattern. It consists of the following conditions:

1. A source model element in the scope of a checkonly domain of a relation, which is set equal to a typed variable, says *x* (line 5).
2. A target model element in the scope of an enforce domain of a relation, which is set equal to the same *x* variable (line 8).
3. The relation R is a top-level relation, or is a non-top-level relation being invoked from a statement of either a *when* or *where* clause of a top-level relation.

So the pattern allows us to get the trace relationship *s→t*. In other words, the pattern says that the value of the target model element will be determined by the value of the source model element.

**Pattern No.2** The use of a function in the enforce domain

To understand our second pattern let us consider a slightly modification in the QVT program presented above. Suppose now we want that the name of the translated instance of M2 be the name of M1 instance plus the suffix "in M2". The Figure 3a shows the new version of relation *classM1_to_classM2*.

In this case we have not only a pivot variable: now the expression of the matching value of the M2 element (line 8) is given by a function F on the variable defined in the checkonly domain (line 5). In this case *F(variable) = variable + 'in M2'*, where operator '+' represents string concatenation operation. The trace relationship obtained in this case will be *F(M1::name)→M2::name*. The Pattern No.2 consists of the following conditions (Figure 3b):

1. A source model element in the scope of a checkonly domain of a relation, which is set equal to a typed variable, says *x* (line 5).
2. A target model element in the scope of an enforce domain of a relation, which is set equal to a function F of the *x* variable (line 8).
3. The relation R is a top-level relation, or is a non-top-level relation being invoked from a statement of either a *when* or *where* clause of a top-level relation.

```
 1. transformation MM1_to_MM2(m1:MM1,m2:MM2) {
 2.   top relation classM1_to_classM2 {
 3.     cname : String;
 4.     checkonly domain m1 a : MM1::M1  {
 5.       name = cname
 6.     };
 7.     enforce domain m2 b : MM2::M2  {
 8.       name = cname + 'in M2'
 9.     };
10.   }
11. }
```

**(a)** The new *classM1_to_classM2* relation

```
 1. transformation T .. {
 2.   top relation R {
 3.     x : VarType;
 4.     checkonly domain source {
 5.       s = x
 6.     };
 7.     enforce domain target {
 8.       t = F(x)
 9.     };
10.   }
11. }
```

**(b)** Pattern No.2

**Fig. 3.** Using a function in an enforced domain.

So, the pattern allow us to get the trace relationship *F(s)→t*. In other words, the pattern says that the value of the target model element will be determined by a function *F* of the value of the source model element. Note that Pattern No.1 is equal to Pattern No.2 when *F(variable) = variable*. So we say that Pattern No.2 is a generalization of Pattern No.1.

**Pattern No.3** The use of a constant value assigned to a target model element

The third pattern is defined for those cases in which a target model element defined in the scope of the enforced domain of a relation is set equal to a constant value. So the content of this element will be always the same, no matter the conditions under which the transformation be executed.

Suppose we modify our M2 class, adding a new attribute *max_length*, meaning that every instance of M2 has a restricted size name, e.g., 30 characters long. So we modified the relation *classM1_to_classM2* to set up this constant value (Figure 4a). The Pattern No.3 then consists of the following conditions (Figure 4b):

1. A target model element in the scope of an enforced domain, which is set equal to a constant *k* (line 8).
2. The relation R is a top-level relation, or is a non-top-level relation being invoked from a statement of either a when or where clause of a top-level relation.

As you can see, no matter what checkonly domain definition was there, this pattern allow us to derive the trace *k→t*.

```
 1. transformation MM1_to_MM2(m1:MM1,m2:MM2) {
```

```
 1. transformation T .. {
```

```
 2.   top relation classM1_to_classM2 {
 3.     cname : String;
 4.     checkonly domain m1 a : MM1::M1  {
 5.       name = cname
 6.     };
 7.     enforce domain m2 b : MM2::M2  {
 8.       name = cname + 'in M2',
 9.       max_length = 30
10.     };
11.   }
12. }
```

**(a)** Setting up a constant in an enforced domain

```
 2.   top relation R {
..
 7.     enforce domain target {
 8.       t = k
 9.     };
10.   }
11. }
```

**(b)** Pattern No.3

**Fig. 4.** Trace inference from a constant value

**Pattern No.4** The use of an variable defined as a function in the *where* clause

Sometimes more complex matching expressions need to be expressed, for example when additional model elements are involved in the relation. These complex expressions must be defined in the where clause of the relation. The where clause specifies the conditions that must be satisfies by all model elements participating in the relation, and it may constrain any of the variables in the relation and its domains. In this case, it is possible to get traceability information as well. The Pattern No.4 stated how to get a trace relationship from an expression in a where clause.

```
 1. transformation MM1_to_MM2(m1:MM1,m2:MM2) {
 2.   top relation classM1_to_classM2 {
 3.     cname : String;
 4.     checkonly domain m1 a : MM1::M1  {
 5.       name = cname
 6.     };
 7.     enforce domain m2 b : MM2::M2  {
 8.       name = cname + 'in M2',
 9.       max_length = 30
10.     };
11.     where {
12.       fullname = cname + 'in M2'
13.     }
14.   }
15. }
```

**(a)** The use of a function in a *where* clause

```
 1. transformation T .. {
 2.   top relation R {
 3.     x, y : VarType;
 4.     checkonly domain source {
 5.       s = x
 6.     };
 7.     enforce domain target {
 8.       t = y
 9.     };
10.     where {
11.       y = F(x);
12.     }
13.   }
14. }
```

**(b)** Pattern No.4

**Fig. 5.** Trace inference from a function in a *where* clause.

Suppose we rewrite our relation *classM1_to_classM2* adding a new string variable *fullname*, which contains the full name of the M2 instance after the transformation finishes (Figure 5a).

In this case, the variable *fullname* acts as a pivot between the attribute *name* (line 8) of the M2 class and the attribute *name* of the M1 class (line 5). The value of the M1 class name is given by the variable *cname* in the expression of the *where* clause (line 12). The Pattern No.4 then consists of the following conditions (Figure 5b):

1. A source model element in the scope of a checkonly domain of a relation R, which is set equal to a typed variable, says *x* (line 5).
2. A target model element in the scope of an enforce domain of the relation R, which is set equal to another typed variable, says *y* (line 8).

3. A statement within the *where* clause of relation R, where the variable *y* is set equal to a function F of the x variable (line 11).
4. The relation R is a top-level relation, or is a non-top-level relation being invoked from a statement of either a when or where clause of a top-level relation.

This pattern allows us to obtain the trace $F(s) \rightarrow t$. Furthermore, the function *F* may include many variables, each of them associated with different source model element. In this case, the derived trace will be $F(s_1,..,s_n) \rightarrow t$.

**Pattern No.5** Trace inference from an *If-Then-Else* statement

The statements used in a QVT-R program has usually the form $x = y$, where *x* can be a variable (e.g. within a *where* clause) or a model element (e.g. in the context of a checkonly or enforce domain), and *y* is an OCL expression. The Object Constraint Language (OCL) [11] is a formal language traditionally used to describe expressions on UML models. In this case, the OCL expressions are used to describe matching expressions in domains, when or where clauses within relations of a QVT program. The OCL language provides a conditional statement *If-Then-Else* as many other languages. A trace relationship can be obtained from this kind of structure. The Pattern No. 5 specifies the conditions to do so.

Suppose we modify our M1 class, adding a new attribute *alias*. So the M1 class has now two properties: a *name* and an *alias*. Then we redefine our *classM1_to_classM2* relation to change the translation rule of an M1 class into an M2 class, meaning that the name of the M2 class be the name of the M1 class, if and only if the name of M1 class is not blank. If so, then the name of the M1 class will be the alias of the M1 class.

Figure 6a shows the new version of *classM1_to_classM2* relation. The name of the M2 class (line 9) is given now by a conditional *If-Then-Else* statement (line 13). In this example two source model elements (*name* and *alias*) are involved, but this is not a mandatory constraint for the pattern. Instead, there could be just a single variable within the *Then* or *Else* expressions, or the same variable in both. The Pattern No.5 then consists of the following conditions (Figure 6b):

1. At least one source model element in the scope of a checkonly domain of a relation R, which is set equal to a typed variable, says *x* (line 5).
2. A target model element in the scope of an enforce domain of the relation R, which is set equal to another typed variable, says *z* (line 9).
3. A statement within the *where* clause of relation R, where the variable *z* is set equal to an *If-Then-Else* statement (line 12).
4. An expression *expr1* in the *Then* condition, and an expression *expr2* in the *Else* condition which contains, at least one of them, a variable defined in the scope of the checkonly domain of R, e.g., variable *x* (line 5) or variable *y* (line 6).
5. The relation R is a top-level relation, or is a non-top-level relation being invoked from a statement of either a when or where clause of a top-level relation.

```
 1. transformation MM1_to_MM2(m1:MM1,m2:MM2) {
 2.   top relation classM1_to_classM2 {
 3.     cname, aname, fullname : String;
 4.     checkonly domain m1 a : MM1::M1  {
 5.       name = cname
 6.       alias = aname
 7.     };
 8.     enforce domain m2 b : MM2::M2  {
 9.       name = fullname
10.       max_length = 30
11.     };
12.     where {
13.       fullname = if cname <> ''
14.                    then cname
15.                    else aname
16.                 endif;
17.     }
18.   }
19. }
```

**(a)** A conditional statement in a *where* clause

```
 1. transformation T .. {
 2.   top relation R {
 3.     x, y, z : VarType;
 4.     checkonly domain source {
 5.       s1 = x
 6.       s2 = y
 7.     };
 8.     enforce domain target {
 9.       t = z
10.     };
11.     where {
12.       z = if condition()
13.          then expr1
14.          else expr2
15.        endif;
16.     }
17.   }
18. }
```

**(b)** Pattern No.5

**Fig. 6.** Trace inference from an *If-Then-Else* statement.

This pattern allows us to obtain the trace *expr1*| *expr2*→t, where *expr1* and *expr2* are expressions that contain one or more references to source model elements. In our example of Figure 6a, the trace obtained is *M1::name*| *M1::alias*→*M2::name*, i.e., the name of the M2 class is given by the name of the M1 class or by the alias of M1. The symbol "|" represents the "OR" logical operation.

We have categorized these kinds of traces as "conditional" traces. A conditional trace is a trace that has been obtained from a conditional expression like an *If-Then-Else* statement, so we can not determine the relationship accurately until transformation finishes. However, a conditional trace shows all the possible relationships among the involved model elements.

**Pattern No.6** Trace inference from a query

One of the features of the QVT language is the possibility of evaluating expressions over a model through queries. A query is just a function that carries out some processing and returns a value. The Pattern No.6 is about getting traceability data from a QVT query.

In our example of Pattern No.3, we define the attribute *max_length* to restrict the size of a M2 class instance name. Now we create a query *N1ToN2()* that receives the name of the M1 class instance and a limit as parameters and returns the name of the M2 class instance. Figure 7a shows the new version of the *classM1_to_classM2* relation and the query *N1ToN2()*. The name of the M2 class instance after translation (line 8) now depends on the M1 class instance name (line 5) and the result of the query *N1ToN2()* defined at line 15.

```
1. transformation MM1_to_MM2(m1:MM1,m2:MM2) {
2.   top relation classM1_to_classM2 {
3.     cname, fullname : String;
4.     checkonly domain m1 a : MM1::M1  {
5.       name = cname
6.     };
7.     enforce domain m2 b : MM2::M2  {
8.       name = fullname
9.       max_length = 30
```

```
1. transformation T .. {
2.   top relation R {
3.     x, y : VarType;
4.     checkonly domain source {
5.       s = x
6.     };
7.     enforce domain target {
8.       t = y
9.     };
10.     };
11.     where {
12.       fullname = N1toN2(cname,max_length);
13.     }
14.   }
15.   query N1toN2(n:String,l:Int):String {
16.     if (n.size() > 1)
17.       then n.substring(1,l)
18.       else n
19.     endif;
20.   }
21. }
```

**(a)** Trace inference from a query

```
10.     where {
11.       y = Q(x);
12.     }
13.   }
14.   query Q(params) : VarType {
15.     ..
16.   }
17. }
```

**(b)** Pattern No.6

**Fig. 7.** Trace inference from a query.

The Pattern No.6 then consists of the following conditions (Figure 7b):

1. At least one source model element in the scope of a checkonly domain of a relation R, which is set equal to a typed variable, says *x* (line 5).
2. A target model element in the scope of an enforce domain of the relation R, which is set equal to another typed variable, says *y* (line 8).
3. A statement within the *where* clause of relation R, where the variable *y* is set equal to a query expression *Q* (line 11).
4. A query *Q* defined in the QVT program which carries out some processing and returns a value (line 14).
5. The relation R is a top-level relation, or is a non-top-level relation being invoked from a statement of either a *when* or *where* clause of a top-level relation.

This pattern allows us to obtain the trace relationship *Q(s)→t*. Although the result of the query *Q* depends on the execution of the transformation, the trace reveals the relationship between both source and target model elements. Considering the QVT program excerpt of Figure 7a, the Pattern No.6 allows us to get the traceability relationship *N1ToN2(M1::name)→M2::name*. So we can derive that *M2::name = N1ToN2(M1::name)*, i.e., the name of the M2 instance will be determined by the processing of the name of the M1 class instance (source), carried out by the query *N1ToN2()*.

## 4 Evaluation

In order to understand the advantages of the technique that we present in this paper, several aspects that differentiate it from other approaches are evaluated. In this section we compare VBA from two different dimensions: the VBA technique versus the QVT dedicated tracing support; and the VBA technique versus other traceability techniques.

### 4.1 The VBA trace vs. the QVT trace

As we said earlier, QVT provides dedicated support for tracing. So after a QVT transformation finishes, a set of obtained traces are recorded, and can be serialized to a post-mortem inspection. The main limitation of the QVT trace is that the information contained in the trace is relative to some kind of elements (*Class* and not *Property*). In the first example of Figure 2, the QVT implicit trace will be *a→b*, indicating that a model MM1::M1 (variable *a* refers to the M1 class of metamodel MM1, and variable *b* refers to the M2 class of metamodel MM2) is related with a model MM2::M2 in some way. Instead, the VBA trace will contain the traceability relationship among the elements (attributes) of the two models involved in the transformation. Following the QVT code excerpt in Figure 2 the VBA trace will be *M1::name→ M2::name*, indicating that the name of the M1 class is related to the name of the M2 class.

### 4.2 The VBA trace vs. other traceability techniques

The automatic generation of traceability information has been the subject of several research papers. One of the first studies of traceability in model transformations can be found in [6]. It is based on traces generation through a loosely coupled process, without altering the definition of model transformations in the context of the ATL language (ATLAS Transformation Language). The attainment of traceability information is performed at the transformation level, so it requires the execution of the transformation to generate the traces. The proposed approach, does not affect the logic of the definition, but adds additional information that blurs the legibility of the original transformation. In contrast, the proposed VBA technique does not pollute either models or transformation specification to get traceability information. Moreover, the VBA approach does not depends on the QVT engine implementation.

Grammel *et al.* [5] proposes a generic traceability framework for augmenting arbitrary model transformation approaches with a traceability mechanism. This generic traceability framework is based on a domain-specific language for traceability (Trace-DSL), presenting the formalization on integration conditions needed for implementing traceability. This work proposes a generic solution for different model transformation languages. This mechanism operates at transformation level, i.e. requires the execution of the transformation to generate the traces, and it allows the interaction with an arbitrary set of model transformation languages, but depends strongly on the connector that enables association to the transformation engine. Our VBA approach does not depend on the transformation execution. Furthermore, it allows obtaining the traceability relationship among the involved models, not being limited to the trace links generation.

In [4], Drivalos *et al.* introduce a dedicated mechanism on the Traceability Metamodelling Language (TML) for detecting and evolving problematic trace links, focusing on the reduction of manual effort incurred in traceability maintenance tasks. In this approach, the traceability relationships are not inferred by a specific automated process. Instead, they are explicitly defined during the model definition.

Aranega *et al.* [2] propose an alternative trace to overcome the limitations of the trace provided by QVT implicit traceability mechanism. This approach, named Local Trace, allows gathering information not provided by the QVT trace, and it is language-independent. Moreover Local Trace can be used without perturbing the QVT transformation execution, or the trace generation/exploitation by the engine.

In [15], Wimmer *et al.* propose a debugging support for the QVT Relation language based on an alternative traceability approach. They convert a QVT Transformation to a colored Petri nets transformation using TROPIC, a model transformation framework based on colored Petri nets. Once again, this mechanism produces data only after transformation execution or simulation.

## 5 Conclusions and future work

In the context of Model-Driven Development traceability is an important topic due to the many applications and benefits it can provide: impact analysis, coverage analysis, and so on.

We have addressed in this work a novel technique to get traceability information from a QVT Relation transformation. The proposed approach is based on the analysis of variables within a QVT-R program to identify certain patterns in the structure of a QVT-R transformation definition. These patterns, in turn, allow us to infer trace relationships among the elements of the models involved in the transformation. Our technique was implemented to be fully automated, and it is completely independent of the transformation execution since it works on the transformation definition. In fact, the VBA approach allows us to decouple the process of obtaining traceability information from the transformation execution.

To evaluate our work, we have compared the QVT implicit trace against our VBA trace. The VBA trace provides enhanced traceability information, including the trace relationship among the elements within the transformation. This could be a key issue in a traceability information high-demand scenario like impact analysis or model synchronization.

The automatic generation of traceability information has been the subject of several research works, we reviewed them. Our variable-based analysis offers a novel approach: no other work is based on code analysis to getting traceability information. This gives us two major benefits: independence from the QVT engine and the possibility to obtain traceability relationship between the models involved in the transformation.

The proposed technique has been implemented as an Eclipse plug-in, and has been tested in two well-known scenarios: UML2Java and BibTeXML2DocBook [8]. Our future research is directed to the identification of new trace patterns that allow us inferring traceability information so far unidentified. Another point of interest for future work is the determination of the whole kind of traces this technique can generate, i.e. the scope of the VBA approach.

## References

1. N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model Traceability. IBM System Journal, 45(3):515–526, 2006.
2. Vincent Aranega, Anne Etien, and Jean-Luc Dekeyser. Using an Alternative Trace for QVT. In Workshop on Multi-Paradigm Modeling, Olso, Norway, October 2010.
3. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. IBM System Journal, 45(3):621-645, July 2006.
4. Nikolaos Drivalos-Matragkas, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. A state-based approach to traceability maintenance. In Proceedings of the 6th ECMFA Traceability Workshop, pages 23-30, NY, USA, 2010.
5. Birgit Grammel and Stefan Kastenholz. A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-Driven Software Development. In Proceedings of the 6th ECMFA Traceability Workshop, pages 7–14, Paris, 2010.
6. Frédéric Jouault. Loosely Coupled Traceability for ATL. In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, pages 29–37, Nuremberg, Germany, November 2005.
7. Anneke G. Kleppe, Jos Warmer, and Wim Bast. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
8. Omar Martínez Grassi. Soporte de trazabilidad en el proceso de transformación de modelos. Master's thesis, Universidad Nacional de La Plata (UNLP), Dec. 2014. http://hdl.handle.net/10915/43494.
9. Object Management Group. MDA Guide Version 1.0.1. http://www.omg.org/cgibin/doc?omg/03-06-01.pdf, June 2003.
10. Object Management Group. Meta object facility (MOF) 2.0 Query/View/Transformation specification version 1.0. http://www.omg.org/spec/QVT/1.0/PDF/, April 2008.
11. Object Management Group. Object Constraint Language - version 2.4. Technical Report formal/2014-02-03, 2014.
12. The Institute of Electrical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology. New York, USA, September 1990.
13. Manuel Wimmer, Angelika Kusel, Johannes Schoenboeck, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets. In Model Driven Engineering Languages and Systems, Lecture Notes in CS, pages 727-732. Springer Berlin, 2009.
14. Stefan Winkler and Jens Pilgrim. A Survey of Traceability in Requirements Engineering and Model-Driven Development. Software System Model. 9(4):529-565, September 2010.